Distributed Diagnosis by Vivid Agents

Michael Schroeder and **Gerd Wagner**
Inst.f. Rechnergest. Wissensverarbeitung, Univ. Hannover, schroeder@kbs.uni-hannover.de
Inst.f.Informatik, Univ. Leipzig, gw@inf.fu-berlin.de

Abstract

Many systems, such as large manufacturing systems, telecommunication networks, or home automation systems, require distributed monitoring and diagnosis. In this article, we introduce a meta-logic interpreter for *vivid agents* which allows to develop distributed monitoring and diagnosis systems consisting of a variety of scalable knowledge- and perception-based agents. The interpreter is based on PVM-Prolog, an extension of standard Prolog with message passing facilities. We show how to specify and run vivid diagnosis agents carrying out fault-tolerant diagnosis of a computer network.

Keywords: Multi Agent Systems, Logic Programming, Diagnosis

Introduction

A multi-agent approach to the diagnosis of distributed systems has several advantages: it achieves fault tolerance without any special hardware, diagnosis agents just need parts of the overall system description which is often too large or not available, and the system can be extended and maintained more easily. Multi-agent diagnosis involves several tasks: agents continuously monitor the operation of the subsystems they are assigned to; if they detect any suspicious behavior they may run specific tests in order to confirm or abandon a certain suspicion; they may communicate their findings to other competent agents, and they may receive requests to run certain tests from other agents; given the findings of other agents they have to compute diagnoses and inform other agents about them; if there is no agreement on the right diagnosis of a malfunction there has to be a mechanism how to resolve conflicts and reach consensus (e.g. by some voting scheme, or by negotiation).

Thus, a diagnosis agent must be capable of performing tests concerning its own state, to compute diagnoses based on other agents' test results and to find a consensus among different diagnosis results. To meet these requirements a diagnosis agent needs an expressive knowledge base that contains a description of the system to be diagnosed and strategic knowledge to deal with hierarchies, multiple models, measurements and preferences. All these requirements are met by *vivid agents*, proposed in (Wagner 1996), which allow various forms of knowledge representation (including the CWA, deduction rules with negation-as-failure, default rules with two kinds of negation), and the rule-based specification of inter-agent cooperation.

The concept of vivid agents comprises both reactive and pro-active behavior competences. The reactive behavior of vivid agents is, however, not hardwired in the form of fixed stimulus-response schemes but encoded in the form of *reaction rules* with a twofold premise: a triggering perception event and an epistemic condition referring to the current knowledge state of the agent. Thus, the reactions of vivid agents may depend on the result of deliberation which is essential for diagnosis agents. At present we do not include pro-active behavior based on planning and plan execution in our diagnosis agents, since they can be specified by means of reaction rules, and do not need planning capabilities.

In our present implementation the knowledge base of an agent has the form of an extended logic program(Gelfond & Lifschitz 1990), thus allowing for the representation of various forms of incomplete knowledge and of default rules. In combination with contradiction removal (Pereira, Alferes, & Aparício 1991), extended logic programs turned out to be sufficiently expressive and computationally tractable to represent and solve many model-based diagnosis problems.

The present paper combines the approach to diagnosis of (Schroeder, de Almeida Móra, & Pereira 1996) with the general multi-agent system model of (Wagner 1996). It is organized as follows. First, we sketch the vivid agent model including the knowledge system of extended logic programs. After presenting PVM (the parallel virtual machine) and PVM-Prolog, we describe an interpreter for vivid diagnosis agents written in PVM-Prolog. We conclude with the discussion of an example of fault-tolerant diagnosis of computer networks which we have implemented as a multi-agent system using our PVM-interpreter for vivid agents and run on a network of four SUN workstations.

Vivid Agents

A *vivid agent* is a software-controlled system whose state is represented by a knowledge base, and whose behavior is represented by means of *action* and *reaction rules*. Following (Shoham 1993), the state of an agent is described in terms of mental qualities, such as beliefs and intentions. The basic functionality of a vivid agent comprises a knowledge system (including an update and an inference operation), and the capability to represent and perform actions in order to be able to generate and execute plans. Since a vivid agent is 'situated' in an environment with which it has to be able to communicate, it also needs the ability to react in response to perception events, and in response to communication events created by the communication acts of other agents.

Reactions may be immediate and independent from the current knowledge state of the agent but they may also depend on the result of deliberation. In any case, they are triggered by events which are not controlled by the agent. A vivid agent without the capability to accept explicit tasks and to solve them by means of planning and plan execution is called *reagent*. The tasks of reagents cannot be assigned in the form of explicit ('see to it that') goals at run time, but have to be encoded in the specification of their reactive behavior at design time.

The concept of vivid agents is not based on a specific logical system for the knowledge-base of an agent. Rather, it allows to choose a suitable knowledge system for each agent individually according to its domain and its tasks. In the case of diagnosis agents, extended logic programs proved to be an appropriate form of the knowledge base of an agent because it is essential for model-based diagnosis to be able to represent negative facts, default rules and constraints.

Vivid Knowledge Systems

The knowledge system of a vivid agent is based on three specific languages: L_{KB} is the set of all admissible knowledge bases, L_{Query} is the query language, and L_{Input} is the set of all admissible inputs, i.e. those formulas representing new information a KB may be updated with. In a diagnosis setting L_{Input} may be $\{test(\_,\_), diagnoses(\_,\_)\}$, where *test* is used to update other agents' test results and *diagnoses* to update the agents' diagnosis results. While the input language defines what the agent can be told (i.e. what it is able to assimilate into its KB), the query language defines what the agent can be asked. Where L is a set of formulas, L^0 denotes its restriction to closed formulas (sentences). Elements of L^0_{Query}, i.e. closed query formulas, are also called *if-queries*.

Definition 1 (Knowledge System) *An abstract knowledge system (see also (Wagner 1995))* $\boldsymbol{K}$ *consists of three languages and two operations: a knowledge representation language* L_{KB}*, a query language* L_{Query}*, an input language* L_{Input}*, an inference relation* $\vdash$*, such that* $X \vdash F$ *holds if* $F \in L^0_{\mathrm{Query}}$ *can be inferred from* $X \in L_{\mathrm{KB}}$*, and an update operation* Upd*, such that the result of updating* $X \in L_{\mathrm{KB}}$ *with* $F \in L^0_{\mathrm{Input}}$ *is the knowledge base* $\mathsf{Upd}(X, F)$*.*

The knowledge system of relational databases, allowing only atomic sentences and requiring complete information, is denoted by $\boldsymbol{A}$ (for $\boldsymbol{A}$tomic). Knowledge systems extending $\boldsymbol{A}$ conservatively are called *vivid*. Positive vivid knowledge systems use a general Closed-World Assumption, whereas general vivid knowledge systems employ specific Closed-World Assumptions (and possibly two kinds of negation). $\boldsymbol{A}$ can be extended to a general vivid knowledge system, called *relational factbases*, by allowing for literals instead of atoms as information units (see below). Further important examples of positive vivid knowledge systems are temporal, fuzzy and disjunctive databases. All these kinds of knowledge bases can be extended to *deductive knowledge bases* by adding deduction rules of the form $F \leftarrow G$ (Wagner 1995).

Factbases and Extended Logic Programs A knowledge base consisting of a consistent set of ground literals (viewed as positive and negative facts) is called a *relational factbase*. In a factbase, the CWA does not in general apply to all predicates, and therefore in the case of a non-CWA predicate, negative information is stored along with positive. This allows to represent predicates for which the KB does not have complete information.

The schema of a factbase stipulates for which predicates the CWA applies by means of a special set *CWRel* of relation symbols. Explicit negative information is represented by means of a *strong* negation $\sim$. For instance, in the factbase

$$\begin{array}{rcl} CWRel & = & \{conn\} \\ X_2 & = & \{conn(s_1, s_2), \sim hi(s_1)\} \end{array}$$

the CWA applies only to the predicate *conn* representing the connection of components, i.e. if it is not positively confirmed that two components are connected we assume that they are not. In contrast to this, the CWA does not apply to *hi*, and we can distinguish the two cases that we have explicitly observed that a switch is not high and that we do not have information about the switch. I.e. $X_2 \vdash \sim hi(s_1)$ means that switch s_1 is observed to be not-high (i.e. low), whereas $X_2 \vdash \neg hi(s_2)$ only expresses that we cannot infer s_2 to be high, which means that it is either not high or that there is no information. As a kind of natural deduction from positive and negative facts an inference relation $\vdash$ between a factbase X and an if-query is defined in the following way:

$$\begin{array}{cll} (\vdash \neg) & X \vdash \neg p(c) & \text{if} \quad p(c) \notin X \\ (\vdash \sim) & X \vdash \sim p(c) & \text{if} \quad \sim p(c) \in X \\ (\vdash \sim_{\text{CWA}}) & X \vdash \sim p(c) & \text{if} \quad p \in CWRel \;\&\; X \vdash \neg p(c) \end{array}$$

where $p(c)$ stands for an atomic sentence with predicate p and constant (tuple) c. $\sim$ and $\neg$ are called *strong* and *weak* negation ($X \vdash \sim F$ implies $X \vdash \neg F$). Compound formulas are treated according to DeMorgan and double negation rules. A factbase X answers an if-query F by yes if $X \vdash F$, by no if $X \vdash \sim F$, and by unknown otherwise. Updates are recency-prefering revisions:

$$\mathsf{Upd}(X, p(c)) := \begin{cases} X \cup \{p(c)\} \text{ if } p \in CWRel \\ X - \{\sim p(c)\} \cup \{p(c)\} \text{ else} \end{cases}$$

$$\mathsf{Upd}(X, \sim p(c)) := \begin{cases} X - \{p(c)\} \text{ if } p \in CWRel \\ X - \{p(c)\} \cup \{\sim p(c)\} \text{ else} \end{cases}$$

The knowledge system of factbases is denoted by $\boldsymbol{F}$. The extension of $\boldsymbol{F}$ by adding deduction rules leads to *extended logic programs* with two kinds of negation.

Definition 2 *Extended Logic Program*
An extended logic program consists of a factbase and a set of rules of the form

$$L_0 \leftarrow L_1, \ldots, L_m, \neg L_{m+1}, \ldots, \neg L_n \;\; (0 \leq m \leq n)$$

where each L_i is a positive or negative fact ($L_i = A|\sim A$, $0 \leq i \leq n$).

Inference in extended logic programs can be defined by the fixpoint semantics of (Pereira & Alferes 1992), but also model-theoretically as preferential entailment based on stable coherent partial models (Herre, Jaspars, & Wagner 1996) corresponding to the *answer sets* of (Gelfond & Lifschitz 1990). In the next section we briefly describe how to employ extended logic programs to solve model-based diagnosis problems.

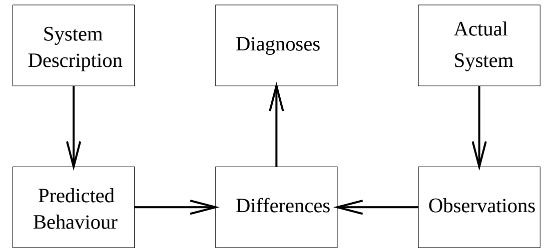

Figure 1: Model-based Diagnosis

Extended Logic Programs and Diagnostic Reasoning

To compute diagnoses (see (Reiter 1987; de Kleer & Williams 1989)) we compare the behavior predicted by a model of the system with the observations of the actual system (see Figure 1). As the predictions are based on the *normality assumption* that all components are working correctly, we can adapt the predictions if they differ from the observations by assuming certain components to be abnormal. We obtain a diagnosis by finding a minimal set of abnormal components which implies a behavior consistent with the observations and, if possible, explaining them.

The description of the system to be diagnosed is formulated as an extended logic program with constraints.

Example 1 *To express the assumption that the system works correctly by default we use negation-as-failure ($\neg$). In the setting of a computer network we may state that we expect component C of unit U to come up with the result CR on test T, whenever CR is the correct result and C is not abnormal:*

$$\begin{array}{l} expected\_result(U, C, T, CR) \leftarrow \\ \quad correct\_result(C, T, CR), \neg abnormal(U, C) \end{array}$$

To express the inconsistency of a disagreement between the predicted behavior and the actual observations, integrity constraints are used.

Definition 3 *Integrity Constraint*
An integrity constraint has the form

$$\perp \leftarrow L_1, \ldots, L_m, \neg L_{m+1}, \ldots, \neg L_n \;\; (0 \leq m \leq n)$$

where each L_i is a literal ($0 \leq i \leq n$), and $\perp$ stands for false.

Example 2 *Now we can express that a contradiction arises if the execution of test T on component C of unit U leads to the result R, and R disagrees with the expected result CR. The second constraint captures that, in the absence of any observed test results, the unit U is in abnormal mode,* fail_stop*, so that no further diagnosis is carried out.*

$$\begin{array}{rcl} \bot & \leftarrow & test(U,C,T), exc(U,C,T,R), \\ & & expected\_result(U,C,T,CR), R \neq CR \\ \bot & \leftarrow & test(U,C,T), \neg exc(\_,\_,\_,\_), \\ & & \neg abnormal(U, fail\_stop) \end{array}$$

In order to restore consistency we change the beliefs that support the contradiction. Those beliefs that are generated by the CWA are preferably changed. They are called *revisables.*

Definition 4 *Revisable*
The revisables R of a program P are a subset of the weakly negated literals which do not have rules in P.

Definition 5 *Revision, Diagnosis*
Let R be the revisables of the program P. A subset $R' \subseteq R$ is called a revision or diagnosis *if it is a minimal set such that $P \cup R'$ is free of contradiction.*

The revision of contradictory extended logic programs is a suitable technique to compute model-based diagnoses.

Specification and Execution of Reagents

Simple vivid agents whose mental state comprises only beliefs, and whose behavior is purely reactive, i.e. not based on any form of planning and plan execution, are called *reagents.* A reagent $\mathcal{A} = \langle X, EQ, RR \rangle$, on the basis of a knowledge system $\boldsymbol{K}$ consists of

(X) a knowledge base $X \in L_{\mathrm{KB}}$,
(EQ) an event queue EQ, being a list of instantiated event expressions, and
(RR) a set RR of *reaction rules*, consisting of epistemic and physical reaction and interaction rules which code the reactive and communicative behavior of the agent.

Operational Semantics of Reaction Rules Reaction rules encode the behavior of vivid agents in response to perception events created by the agent's perception subsystems, and to communication events created by communication acts of other agents. We distinguish between epistemic, physical and communicative reaction rules, and call the latter *interaction rules.* We use L_{PEvt} and L_{CEvt} to denote the perception and communication event languages, and $L_{\mathrm{Evt}} = L_{\mathrm{PEvt}} \cup L_{\mathrm{CEvt}}$. The following table describes the different formats of epistemic, physical and communicative reaction rules:

$$\begin{array}{rcl} Eff & \leftarrow & \mathsf{recvMsg}[\varepsilon(U), S],\ Cond \\ \mathsf{do}(\alpha(V)),\ Eff & \leftarrow & \mathsf{recvMsg}[\varepsilon(U), S],\ Cond \\ \mathsf{sendMsg}[\eta(V), R],\ Eff & \leftarrow & \mathsf{recvMsg}[\varepsilon(U), S],\ Cond \end{array}$$

The event condition $\mathsf{recvMsg}[\varepsilon(U), S]$ is a test whether the event queue of the agent contains a message of the form $\varepsilon(U)$ sent by some perception subsystem of the agent or by another agent identified by S, where $\varepsilon \in L_{\mathrm{Evt}}$ represents a perception or a communication event type, and U is a suitable list of parameters. The epistemic condition $Cond \in L_{\mathrm{Query}}$ refers to the current knowledge state, and the epistemic effect $Eff \in L_{\mathrm{Input}}$ specifies an update of the current knowledge state.

Physical Reaction: $\mathsf{do}(\alpha(V))$ calls a procedure realizing the action α with parameters V.

Communicative Reaction: $\mathsf{sendMsg}[\eta(V), R]$ sends the message $\eta \in L_{\mathrm{CEvt}}$ with parameters V to the receiver R.

Both perception and communication events are represented by incoming messages.

In general, reactions are based both on perception and on knowledge. Immediate reactions do not allow for deliberation. They are represented by rules with an empty epistemic premise, i.e. $Cond = \mathsf{true}$. Timely reactions can be achieved by guaranteeing fast response times for checking the precondition of a reaction rule. This will be the case, for instance, if the precondition can be checked by simple table look-up (such as in relational databases or factbases).

Reaction rules are triggered by events. The agent interpreter continuously checks the event queue of the agent. If there is a new event message, it is matched with the event condition of all reaction rules, and the epistemic conditions of those rules matching the event are evaluated. If they are satisfiable in the current knowledge base, all free variables in the rules are instantiated accordingly resulting in a set of triggered actions with associated epistemic effects. All these actions are then executed, leading to physical actions and to sending messages to other agents, and their epistemic effects are assimilated into the current knowledge base.

A PVM-Prolog-Based Interpreter for Vivid Agents

PVM - the parallel virtual machine - (Geist & et al. 1994) is a standard software in distributed and parallel computing that permits a heterogeneous collection of UNIX computers (ranging from a personal computer or workstation to massively parallel processors) to be viewed as a single parallel computer. PVM coordinates the different hardware architectures and data formats within a network. The PVM computing model is simple, yet very general: tasks are spawned in the network accessing PVM resources through a library of standard interface routines. These routines allow the initiation and termination of tasks across the network as well as communication and synchronization of the tasks.

Besides the standard interfaces to C, C++ and Fortran, PVM is available in Lisp, Ada and Prolog. Its widespread use and generality makes PVM an excellent candidate for implementing interpreters for multi-agent programming languages. A general scheme for such implementations contains PVM as a layer on top of low-level network protocols (see Figure 2). The next layer contains programming languages with full PVM access. On top of them, agent interpreters can be implemented, allowing to run high-level agent programs as the topmost layer. Alternatively, agents can be programmed directly in the lower level of standard programming languages with PVM access. Notice, however, that we rather recommend to use a high-level declarative agent programming language such as our vivid agent specification language in order to achieve readable and compact source code.

Applications					
Agent Interpreter					
C	Fortran	Prolog	Lisp	C++	Ada
PVM					
TCP	UDP				

Figure 2: Layers

PVM and PVM-Prolog

PVM realizes communication of spawned tasks by message-passing. The message-passing primitives are tailored for heterogeneous operation, involving strongly typed constructs for buffering and transmission. Communication constructs include those for sending and receiving data structures as well as high-level primitives such as broadcast, barrier synchronization, and global sum. There are no limits for messages: A task can send an unlimited number of messages of unlimited size, though the size is, of course, limited by the available memory of the host.

An essential feature for multi-agent applications is asynchronous communication. Tasks can perform an asynchronous blocking send and receive. A *blocking* send or receive waits until the message is sent or an incoming message arrives. Blocking is often useful to let the agent idle and safe computational power. Besides the primitive send, a multi-cast to a set of tasks and broadcast to user-defined groups is supported. In general, the message order is preserved in point-to-point communication. Additionally, PVM provides dynamic hardware configuration, i.e. addition and deletion of hosts at run-time and signaling, such that on events such as exit, addition or deletion of tasks other tasks are informed.

All these PVM-primitives are included in PVM-Prolog, a Prolog-core extended by a PVM-interface (Cunha & Marques 1997). In addition to PVM's coarse grain parallelism, PVM-Prolog provides process-internal threads. While PVM tasks run physically distributed the threads are process internal and therefore have less overhead.

An Interpreter for Vivid Agents

To set up the multi-agent system a corresponding number of processes is spawned at different locations each loading a file containing its behavior specification. Once the processes are set up they can communicate through the PVM-primitives `pvm_send` and `pvm_recv`.

```
cycle :-
  newEvt( Evt),
  findall(ActEff, (reaction(ActEff,Evt,Cond),
                      demo(Cond)), ActEffs),
  perform( ActEffs),
  cycle.
newEvt( Msg/From) :-
  pvm_nrecv( -1,1,Msg/From).
newEvt( noEvt).
```

Figure 3: Perception-Reaction-Cycle.

The execution of a reagent is specified by a *perception-reaction cycle* similar to the 'observation-thought-action' cycle of (Kowalski 1995) and the interpreter cycle of AGENT0 (Shoham 1993). First, we check if there is a message (`newEvt`, see Figure 3). If there is one, the predicate `pvm_nrecv`, which realizes a non-blocking receive, succeeds and the message is popped from the event queue. Next, the reaction rules are evaluated. For all reaction rules matching the event expression the precondition `Cond` is tested and in case it holds, the triggered actions are collected in the set `ActEffs`. Subsequently, they are performed and the cycle is closed.

Triggered actions are performed as follows (see Figure 4): The sendMsg-action is translated into PVM-Prolog's `pvm_send`, whereas a physical action do(*Act*) causes a call to a procedure with the same name. The epistemic effects of actions are assimilated into the knowledge base by means of the Prolog operations *assert* and *retract*.

```
perform( [noAct/Effs|ActEffs]) :-
  assimilate( Effs), perform( ActEffs).
perform( [sendMsg(Msg,To)/Effs|ActEffs]) :-
  name2id( To,Id), i_am( Self),
  pvm_send( Id,1,Msg/Self),
  assimilate( Effs), perform( ActEffs).
perform( [do(Act)/Effs|ActEffs]) :-
  call( Act),
  assimilate( Effs), perform( ActEffs).
perform( [halt/_|ActEffs]) :-
  perform( ActEffs),
  pvm_exit, fail.
assimilate( [not(L)|Effs]) :-
  retractall( fact(L)),
  assimilate( Effs).
assimilate( [L|Effs]) :-
  assert( fact(L)),
  assimilate( Effs).
```

Figure 4: Executing Reactions.

Distributed Diagnosis of a Computer Network

To achieve fault-tolerant diagnosis three agents a, b, c diagnose a node d. Since a, b, c have the same description of d their findings are redundant in case they are up and they received the same test results of d. The tasks of the agents can be divided into four steps as shown in Figure 5.

Initially, the agents receive a *start* message from the creator process. In a second step, the diagnosing agents send requests for test results to the group of agents whose underlying hardware is examined, the tests are executed and the requests are answered. Thirdly, the diagnosing agents compute diagnoses based on the new test results they received and communicate their diagnoses among the diagnosis group. Finally, each diagnosis agent computes a consensus based on all the diagnoses it received, and sends it to the (possibly human) agent it has to report to.

Specification of the Diagnosis Agents

This agent behavior is specified declaratively via the following reaction rules based on the communication events

$$CEvt = \{ ask(Query), reply(Answer), tell(Input), request(Action) \}$$

When the creator process tells a diagnoser to start, the diagnoser asks the test agents for a list of all current test results:

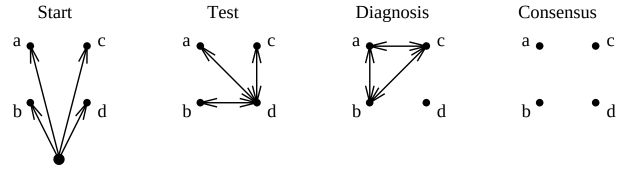

Figure 5: Steps of the Diagnosis Process

$$\begin{aligned}&\mathsf{sendMsg}(ask(testResults), T) \longleftarrow\\ &\quad \mathsf{recvMsg}(tell(start), creator),\\ &\quad i\_am(A), diagnoser(A), tester(T).\end{aligned}$$

When a tester receives a query for test results it performs the tests and replies to the query. To obtain test results low-level system calls can be integrated into the predicate *perform_tests* by procedural attachment.

$$\begin{aligned}&\mathsf{sendMsg}(reply(test(Results)), D) \longleftarrow\\ &\quad \mathsf{recvMsg}(ask(testResults), D),\\ &\quad perform\_test(Results).\end{aligned}$$

If an agent receives a reply with new test results, the agent enters the third phase of computing and exchanging diagnoses among the group of diagnosers. The next reaction rule expresses that an agent which received test results, computes diagnoses based on the received test data and sends the diagnoses to all diagnosers B. Similar to *perform_tests* the predicate *compute_diags* is implemented by procedural attachment. A call of *compute_diags* adds the new data *Results* to the KB and starts a revision of KB, which may be partially inconsistent due to the newly received test data and the agent's default predictions concerning the correctness of involved hardware. The repair of the partial inconsistency yields diagnoses which are returned in the parameter Ds.

$$\begin{aligned}&\mathsf{sendMsg}(tell(diagnoses(Ds)), A) \longleftarrow\\ &\quad \mathsf{recvMsg}(reply(test(Results)), T),\\ &\quad compute\_diags(Ds, Results),\\ &\quad diagnoser(A).\end{aligned}$$

Incoming diagnosis results are added to the knowledge base:

$$\begin{aligned}&diagnoses(A, Ds) \longleftarrow\\ &\quad \mathsf{recvMsg}(tell(diagnoses(Ds)), A).\end{aligned}$$

The incoming diagnosis results are counted and if the last message is received the agent triggers the final phase of consensus computation. The pending diagnosis results are counted by checking if there is a diagnoser for which no diagnosis results are stored. The last result is received if the agent is still waiting for at least one result, but not for two anymore. This is expressed by the following deduction rules:

$$\begin{array}{rcl} wait\_for(A) & \leftarrow & diagnoser(A), \neg diagnoses(A, \_). \\ wait\_for\_one & \leftarrow & wait\_for(A). \\ wait\_for\_two & \leftarrow & wait\_for(A), \\ & & wait\_for(B), A \neq B. \\ last & \leftarrow & wait\_for\_one, \neg wait\_for\_two. \end{array}$$

In case a diagnosis result is received and it is the last one, the agent triggers its voting procedure (by sending a corresponding request to itself):

$$\begin{array}{l} \mathsf{sendMsg}(request(vote), B) \longleftarrow \\ \quad \mathsf{recvMsg}(tell(diagnoses(Ds)), A), \\ \quad i\_am(B), last. \end{array}$$

The final step of computing a consensus does not involve any communication with other agents. On receipt of a request to vote, the agent computes a consensus by calling *vote*, outputs the result and as an effect cleans up its knowledge base from all diagnosis results stored intermediately. The computation of the consensus is procedurally attached. For the purpose of fault tolerance a consensus can be implemented by a simple voting mechanism, i.e. the call *vote* checks all the collected *diagnoses* facts and determines the most frequent result.

$$\begin{array}{l} \neg diagnoses(A, Ds) \longleftarrow \\ \quad \mathsf{recvMsg}(request(vote), A), \\ \quad vote(C), write(C), diagnoses(A, Ds). \end{array}$$

Evaluation: Vivid Agents at Work

A vivid agent for the diagnosis of a computer network is specified by its knowledge base, its reaction rules and a set of specific predicates realizing synchronous actions by means of procedural attachment. In our example the knowledge base comprises knowledge about the agent's own name (*i_am/1*), the agent's classification as diagnoser (*diagnosers/1*) or tester (*testers/1*), rules to compute the expected results and to constrain the difference between expected and actual test results (see section) and temporary facts about tests and diagnoses. The agent's behavior is specified by its reaction rules. In addition, it may perform synchronous actions to run certain tests (`perform_tests/1`), to compute diagnoses (`comp_diags/2`), and to vote for a final diagnostic decision (`vote/1`).

When executed, the above agent specification invoked for four agents a, b, c, d leads to a trace as shown in Figure 6. The agents a, b, c form the diagnosing group which takes care of a tester group with a single member d. As we want to focus on the communication among the agents we have d always sending the same test results. When the agents perform the four steps of starting, testing, diagnosing and computing a consensus, as sketched in Figure 5, they are not synchronized and the steps may overlap. Therefore we represent *send* and *receive* by a bold and dotted arc, respectively. For example, the first icon (1) indicates that b received from the creator a start message and the second (2) means that b sends to d a request message. The receipt of this request message (4) is preceded by d receiving a start message from the creator (3). In general, the trace reveals that agent b is much faster than a and c. For example, after c receives the start message (10) agent b already received test results, computed its diagnoses and starts to broadcast them (11,13,16).

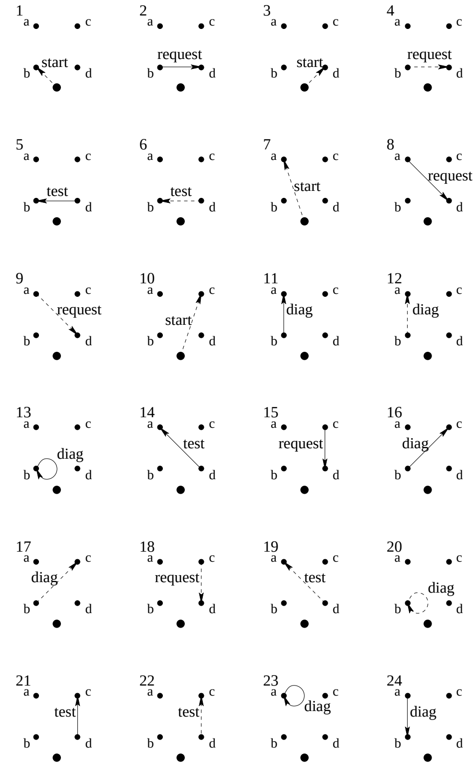

Figure 6: A Trace of Communication of Agents a, b, c diagnosing d. The big node is the creator process.

This small setting of four agents already reveals a central problem: The dynamic behavior of the multi agent system is far from being easily understandable and agent specifications are, due to the complex interactions with other agents, likely to be error-prone so that the formal and automated verification of agent specifications seem to be important for the success of this technology.

Related Work

The ARCHON project (Jennings *et al.* 1996) involves several real-world applications of distributed diagnosis. A main contribution of the ARCHON archictecture is the linkage of several indepedently operating systems to form a cooperating MAS. In contrast to our approach ARCHON is application driven and does not provide a uniform MAS programming environment underpinned by a formal semantics.

There are a number of proposals for high-level agent-oriented programming languages (Shoham 1993; Thomas 1994; Rao 1996). In these languages, the knowledge base of an agent admits only of literals upon which classical inference is performed, i.e. the CWA and negation-as-failure are not used. In the vivid agent approach, however, we take into account that, based on the experience with SQL and Prolog, the CWA and negation-as-failure are essential for information and knowledge processing. Also, neither of these languages allows intensional predicates expressing causal relations and generic laws in the form of possibly non-monotonic deduction rules which are essential for model-based diagnosis.

Conclusion and Future Work

We have shown how the concept of vivid agents can be used to realize a multi-agent system for distributed diagnosis. Our main contribution is the implementation and evaluation of a first prototype for vivid agents based on PVM. The prototype and its application to distributed diagnosis are based on (Schroeder, de Almeida Móra, & Pereira 1996).

Our implementation realizes the fundamental part of the general concept of vivid agents. We plan to use our experiences for the design of a full interpreter for vivid agents. The main issue in this ongoing project is to integrate planning facilities and pro-active behavior into our current prototype. Our preliminary experience suggests that PVM-Prolog can be used as a basis for multi-agent systems: it allows to combine heterogeneous agents implemented in different languages. We are currently implementing an environment in PVM-Ada that allows to spawn both Ada and Prolog agents.

Similar to ACTOR-like approaches, a PVM-based system allows to create new agents at runtime within the local network, but it does not support an open architecture where processes started somewhere else may join the multi agent system. This seems to be a severe limitation which has to be overcome. A possible solution to this problem could be the implementation of an appropriate e-mail communication interface between the PVM-based multi-agent system and external agents.

Acknowledgments We would like to thank Wolfgang Nejdl for his comments and suggestions on an earlier draft of this paper and Iara de Almeida Móra and Rui Marques for many fruitful discussions.

References

Cunha, J. C., and Marques, R. F. P. 1997. Distributed algorithm development with PVM-Prolog. In *5th Euromicro, Ws. on Parallel and Distributed Processing*. London, UK: IEEE Press.

de Kleer, J., and Williams, B. C. 1989. Diagnosis with behavioral modes. In *Proc. of IJCAI89*, 1324–1330. Morgan Kaufmann.

Geist, A., and et al. 1994. *PVM: Parallel Virtual Machine*. MIT Press.

Gelfond, M., and Lifschitz, V. 1990. Logic programs with classical negation. In *Proc. of ICLP90*, 579–597. MIT Press.

Herre, H.; Jaspars, J.; and Wagner, G. 1996. Partial logics with two kinds of negation as a foundation of knowledge-based reasoning. In Gabbay, D., and Wansing, H., eds., *What is Negation?* Oxford University Press.

Jennings, N. R.; Corera, J. M.; Laresgoiti, I.; Mamdani, E. H.; Perriollat, F.; Skarek, P.; and Varga, L. Z. 1996. Using ARCHON to develop real-world DAI applications for electricity transportation management and particle accelerator control. *IEEE Expert*.

Kowalski, R. A. 1995. Using meta-logic to reconcile reactive with rational agents. In *Meta-logics and Logic Programming*, 227–242. MIT Press.

Pereira, L. M., and Alferes, J. J. 1992. Well founded semantics for logic programs with explicit negation. In *Proc. of ECAI92*. 102–106. John Wiley & Sons.

Pereira, L. M.; Alferes, J. J.; and Aparício, J. 1991. Contradiction Removal within Well Founded Semantics. In *Logic Programming and Nonmonotonic Reasoning*, 105–119. MIT Press.

Rao, A. S. 1996. Agentspeak(l): BDI agents speak out in a logical computable language. In *Proc. of MAAMAW96, LNAI 1038*. Springer-Verlag.

Reiter, R. 1987. A theory of diagnosis from first principles. *Artificial Intelligence* 32(1):57–96.

Schroeder, M.; de Almeida Móra, I.; and Pereira, L. M. 1996. A deliberative and reactive diagnosis agent based on logic programming. In *Proc. of ATAL96*. Springer–Verlag. To appear.

Shoham, Y. 1993. Agent-oriented programming. *Artificial Intelligence* 60(1):51–92.

Thomas, S. R. 1994. The PLACA agent programming language. In *Proc. of ATAL94, LNAI 890*. Springer–Verlag.

Wagner, G. 1995. From information systems to knowledge systems. In *Information System Concepts*. Chapman & Hall.

Wagner, G. 1996. A logical and operational model of scalable knowledge-and perception-based agents. In *Proc. of MAAMAW96, LNAI 1038*. Springer-Verlag.